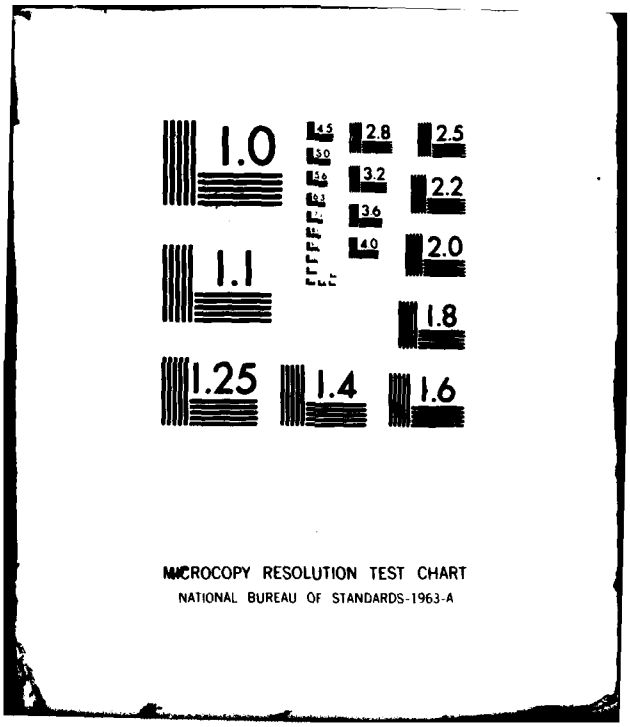
AD-A082 545 STANFORD UNIV CALIF STANFORD ELECTRONICS LABS /6 9/2
DESIGN AND VERIFICATION OF RELIABLE SOFTWARE.(U)
FEB 80 S S OWICKI F49620-77-C 045
UNCLASSIFIED AFOSR-TR-80-0232 NL

END
DATE FILMED
5 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

COMPUTER SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING

STANFORD UNIVERSITY

Stanford, California 94305

DESIGN AND VERIFICATION OF RELIABLE SOFTWARE

FINAL REPORT

Covering the Period 1 January 1977 - 31 December 1979

AFOSR Contract no. F49620-77-C-0045

SEL Project N-771

Prepared for the

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

Bolling Air Force Base
Washington, D. C. 20332

Principal Investigator:

Professor Susan S. Owicki

February 1980

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-80-0232	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): DESIGN AND VERIFICATION OF RELIABLE SOFTWARE	5. TYPE OF REPORT & PERIOD COVERED: Final, 1 Jan 77 - 31 Dec 79
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Susan Owicki	8. CONTRACT OR GRANT NUMBER(s): F49620-77-C-0045
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Stanford University, Computer Systems Laboratory, Stanford, CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, Washington, D.C. 20332	12. REPORT DATE: Feb 1980 / 13. NUMBER OF PAGES: 11
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program verification, operating system design, parallel programming, concurrent programming, program proving, temporal logic, distributed databases, network protocols.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This research project is concerned with methods for developing provably correct concurrent programs. Two complementary approaches have been pursued. The first is the development of basic tools for describing and verifying concurrent interactions between system components. Specification and proof methods have been developed for both invariant properties and service guarantees of modular programs. They allow independent verification of each module, and provide a basis for reasoning about a wide class of concurrent systems. Statement and verification of service guarantees is made possible by using temporal logic, a logica

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

382400

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

20. Abstract cont.

logical system in which one can reason about the future states of a program computation.

The second approach is the investigation of specific problems in three application areas: operating systems, network communications protocols, and distributed databases. In addition to providing test cases for evaluating the basic tools, these applications are important in their own right. It has been possible to identify common patterns, and to build more sophisticated techniques for handling them from the basic tools discussed above. A major issue in certain applications is dealing with reliable components, and methods of reasoning about such components are presented.

TABLE OF CONTENTS

1.	Research Objectives	1
2.	Description of Research and Results	1
3.	Publications	7
4.	Professional Personnel	8
5.	Interactions	9

1. RESEARCH OBJECTIVES

Program verification has frequently been suggested as a tool for improving software reliability. The purpose of this research has been to develop verification techniques that can be applied to complex program systems involving concurrency, such as operating systems and network communications managers.

Two complementary approaches have been pursued. The first is the development of basic tools for describing and verifying concurrent programs; the aim is to find ways of reducing the complexity that arises from concurrent interactions between the components of the system. These tools are general purpose and should be applicable to a wide range of problems. The second approach is the investigation of specific applications by analyzing algorithms and programs from such domains as operating sytems, networks, and distributed databases. These program examples provide test cases for evaluating the power of the basic tools and help to identify problem areas where further basic work is needed. In addition, analysis of these programs often makes it possible to recognize common patterns and formulate rules that simplify the design and verification of program components that fit these patterns.

2. DESCRIPTION OF RESEARCH AND RESULTS

2.1 Basic Tools

Our purpose here has been to devise general-purpose specification and proof methods for concurrent programs. Two types of correctness criteria can be distinguished: invariant properties, which should be true throughout system execution, and liveness constraints, which require that certain events must eventually occur. For example, an airline reservation system might preserve the invariant that no flight is overbooked and provide a liveness guarantee that each request for a reservation is eventually answered. Our techniques for dealing with the two kinds of problems are discussed below. In both cases, we are concerned with managing the complexity of the verification process, and so favor modular methods that allow factoring the verification of a system, with relatively independent treatment of each component.

First, let us consider the proof of invariant properties. Concurrent programs can be constructed using three kinds of modules: processes (the active components), monitors (which implement shared data objects and operations), and compound modules constructed from other modules. We have developed a specification format in which each module is described by assertions giving its initial state, its requirements from other modules, invariant relations between its local variables, and effects of its procedures (if it has any). Verification rules for proving that a module implementation meets its specifications take several forms. For process and monitor modules, verification requires direct analysis of the program code. Because of the specification style, this step is in general no more complex than for a non-concurrent program. For compound modules,

verification depends only on the specifications of the components, and not on their implementaions. Since each module verification is performed independently, and the effect of interactions between components is limited to a small, well-defined interface, the complexity of the system proof does not grow unmanageably as the size of the system increases.

The basic rules for proving invariant properties of modular systems are presented in [1]. In [2], the rules are amplified and illustrated by the design and verification of a simple system for routing mail in a ring network.

2.1 Liveness

The methods of specifying and verifying invariant properties described above are adequate for a wide range of concurrent programs. Our work on liveness properties is at an earlier stage, but the initial results are promising. Our approach is based on temporal logic, in which one can express assertions about future program states. This is accomplished by introducing two symbols: $\Diamond$ (pronounced eventually) and $\Box$ (pronounced henceforth). The liveness requirements of the airline reservations system described above can be expressed in temporal logic by:

$$\Box(\textit{if } (\textit{request } R \textit{ received}) \textit{ then } \Diamond(\textit{request } R \textit{ answered})).$$

In earlier work, the statement and verification of liveness properties was done informally. As a result, correctness proofs were very susceptible to ambiguities in the problem statement and to errors of reasoning. The main contribution of temporal logic is that it provides a precise means of stating and proving liveness requirements.

To facilitate modular proofs of liveness, the specification methods for modules described in the previous section have been extended to include liveness properties. These properties are expressed as conditional promises with the form

$$\textit{if } P \textit{ then } \Diamond Q \textit{ unless delayed by } D,$$

where P, Q, and D are assertions about the program state. The meaning of such a promise is that if P is true of some state in a computation, then there must be a subsequent state in which Q is true, except in cases where D remains true forever.

As an example, consider a pipeline of processes communicating through buffers. Process P_i, which moves data from buffer B_{i-1} to buffer B_i, might have the liveness specification:

$$\textit{if } (x \textit{ is in } B_{i-1}) \textit{ then } \Diamond (x \textit{ is not in } B_i \textit{ and } f_i(x) \textit{ is in } B_i) \textit{ unless delayed by } (B_i \textit{ full}).$$

Informally, this states that process P_i promises to move items from its input buffer to its output buffer unless its output buffer stays full.

Verifying that a module implemented by a process or monitor meets its liveness promises depends on axioms that characterize the liveness properties of programming

language statements. (The properties of synchronizing statements, like semaphore P and V operations and monitor waits, are especially important). For compound modules, the liveness of the whole module can be verified using just the specifications of its components. For example, in the pipeline program, the entire system of processes and buffers satisfies the unconditional promise

$$if\ (x\ is in\ B_0)\ then \diamond (f_k\ (f_{k-1} \ldots (f_1(x)\ldots))\ is\ in\ B_k)$$

The proof of this unconditional system promise involves showing that P_i's promise to remove items from B_{i-1} cancels out P_{i-1}'s delay condition. Proof rules for verifying promises of this sort have been developed for certain applications, notably the protocol verification problems discussed in Section 3.2.

2.2 Applications

We have investigated the design and verification of concurrent systems in three applications areas: operating systems, network communications protocols, and distributed databases. The work in each area is described below.

Operating Systems

We have considered the design of an operating system nucleus, identification of common patterns for operating system modules, and schemes for the detection and prevention of deadlock.

Two alternatives for organizing the nucleus were compared and evaluated. In the first, message passing, the program consists of a set of independent processes. The processes have no common memory or variables, but can communicate by sending and receiving messages. In the second method, processes may share memory, but each set of shared variables is encapsulated in a module. The module also contains procedures; a process can only access module data by calling module procedures. Modules provide protection from time-dependent errors, since only one process at a time may be active in a module.

In order to compare these alternatives, we prepared a detailed design of the nucleus memory manager using each method and compared their ease of verification. Our conclusion was that it was easier to create and verify the module-based design. This is primarily because of the convenience of shared data, and because the procedure call used with modules imposes more structure on the relationships between system components. For this reason, we chose to organize our design according to the data-module model.

The use of data-modules as a basic structuring tool simplifies verification of most of the nucleus. However, the kernel, which provides the implementation of modules, is more complicated than a kernel for implementing message passing. To compare them, we developed high-level designs for both types of kernel. While the module-based kernel was indeed more complicated, the difference was not great enough to outweigh the advantages of using modules in the rest of the nucleus.

An important result of our study of the nucleus was the identification of two patterns that account for most of the modules of an operating system. One is the *transmitter*, which produces a stream of output values from a stream of input values. Verifying properties of transmitter modules is based on *history sequences* that record the sequences of input and output values. An example of verification of a system composed of transmitter modules is given in [2]. The other common pattern is the *dynamic resource allocator*, which manages the sharing of some object(s) between competing modules. A typical example of such an allocator is a memory manager that maps program pages to memory pageframes. The modular proof techniques we favor gain much of their power from the fact that shared data is statically allocted to one module, so dynamic allocation is a complicating factor. However, it is possible to specify a method of dynamic allocation, based on capabilities, that fits the modular verification style. In [3], we precisely specify this method, and show how to verify that an allocator implementation meets its requirements. Once the allocator itself has been verified, the correctness proof of the rest of the system is no more difficult than if allocation were strictly static.

Deadlock avoidance is a particular problem for resource allocators, and we have investigated it in some detail. A variety of deadlock-avoidance strategies have been proposed in the literature. They have in common the property that deadlock is prevented by refusing to allocate a resource if doing so leads to an "unsafe" state: one in which further resource requests might result in deadlock. The strategies differ in the amount of computation they perform in evaluating a request and in the amount of information they require about future resource requests from the competing modules. In general, a strategy that performs more computation or uses more information about resource needs can recognize a larger number of requests as "safe," and so impose less delay on the competing modules. We have developed a model that allows a precise characterization and comparison of these differences; analysis of the model has suggested extensions that improve the performance of the previous algorithms. This work is reported in [5].

Network Communication Protocols

In network communications protocols, most of the modules are instances of the same transmitter pattern that occurs so often in operating systems. The principle difference is that communication protocols must function correctly even in the presence of processor failures and transmission line errors. We can incorporate these failure possibilities into the verification in a straightforward manner. For example, the invariant of a perfect communication link is that the ouput it has delivered is an initial segment of the input it has received. A communication link that may lose or re-order messages is characterized by an invariant that states that its output is a permutation of a subsequence of its input. The liveness promise of a failure-free link is that each input value will eventually be output, while the promise of a link that can lose a message a bounded number of times is that any message that appears a sufficient number of times in the input will eventually be output.

We have found that the modular techniques already discussed are adequate for proving invariant properties and liveness of several types of data-transmission protocols.

The proof of invariant properties is a straightforward extension of the kind of reasoning used with transmitters in operating systems. Novel features include the possibility for loss and re-ordering allowed by the invariant for the transmission medium, and the cycles in the data paths that occur because of the need to acknowledge transmissions. These features cause no theoretical difficulties, but they seem to lead to more complex module invariants.

In proving liveness, the principal issue is that actions such as transmitting messages and acknowledgments must be repeated until they are successful. For example, a transmitter may have the liveness property that it promises to keep sending message n until it receives acknowledgment n. This can be expressed in temporal logic by

$$if\ \Box(acknowlegement\ n\ not\ received)\ then\ \Box\Diamond(message\ n\ sent.)$$

The corresponding receiver may have the liveness promise that it will repeatedly acknowledge message n until it receives message $n+1$.

Distributed Databases

The final application area we have considered is distributed databases. Here the issue of concern is maintaining the consistency of multiple copies of data while allowing access from several users to take place concurrently. A new consistency control algorithm has been developed for this purpose [5]: it uses a distinguished *true copy* of each data item which is the locus of locking for that data item. The true copy may migrate throughout the system, and may be split into multiple *shared copies* that can be read but not modified. This allows concurrent operation, with a lower overhead of messages than many existing algorithms. The consistency of the true-copy algorithm can be verified using existing techniques: consistency is another example of an invariant property. The true-copy algorithm is independent of any particular policy for avoiding deadlocks or resolving competing resource demands, so it can serve as a basis for a variety of schemes whose consistency will then be easily verified.

A further application of the true-copy mechanism is its use to provide resilient system operation when some of the system components fail. For example, when a processor managing certain data items fails, a resilient system should be able to continue operation using other copies of the data items. However, this must be done in such a way that system consistency is maintained, in spite of the fact that transactions at the failed site may already have modified the local copies of the data. Several mechanisms for providing resilient systems are discussed in [6].

2.3 Summary

Our experience in studying application problems leads to the following assessment of the state of verification techniques for concurrent programs. The methods of proving invariant properties are well-understood, and are adequately powerful for a wide range of systems. Further work in deriving techniques tailored to particularly common patterns,

like the transmitter and allocator already discussed, would still be profitable. Proof of liveness properties is at an earlier stage, but we have been able to deal with a number of sample problems. Proofs of network protocols, which must cope with unreliability, are especially encouraging. Further work is needed to identify common liveness patterns, like the ones described for invariants, and to provide techniques for verifying systems that follow these patterns.

3. PUBLICATIONS

[1] Owicki, S. "Specifications and Proofs for Abstract Data Types in Concurrent Programs," in Bauer and Broy (ed.), *Program Construction*, Springer-Verlag, 1979.

[2] Owicki, S. "Specification and Verification of a Network Mail System," in Bauer and Broy (ed.), *Program Construction*, Springer-Verlag, 1979.

[3] Owicki, S. "Verifying Parallel Programs with Resource Allocation," *Proceedings of the International Conference on Mathematical Studies of Information Processing*, Springer-Verlag, 1979.

[4] Minoura, T. "A New Concurrency Control Algorithm for Distributed Database Systems," *Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks*, August, 1979.

[5] Minoura, T. "Deadlock Avoidance Revisited," accepted by *Journal of the ACM*.

[6] Minoura, T. Resilient Extended True-Copy Token Algorithms for Distributed Database Systems, Ph. D. thesis, Stanford University, to appear June 1980.

4. PROFESSIONAL PERSONNEL

Principle Investigator: Susan Owicki

Graduate Student Research Assistants:

Keith Marzullo, 9/77 to present

Toshimi Minoura, 9/77 - 1/78 and 1/78 - present

Alfred Spector, 1/77 - 6/77

5. INTERACTIONS

5.1 Spoken Papers

Owicki, S. Lecturer at NATO International Summer School on Program Construction, Germany, July, 1978.

Owicki, S. "Modelling Parallel Programs Using Temporal Logic," IFIP Working Group 2.2, Kyoto, Japan, August 1978.

Owicki, S. "Verifying Protocols as Parallel Programs," at Protocol Verification Workshop, March 20-21, 1979. Sponsored by DARPA, organized by Vint Cerf.

Owicki, S. "Safe Garbage Collection in the Presence of Concurrency," IFIP Working Group 2.3, Santa Cruz, California, August 1979.

5.2 Other Interactions

Susan Owicki is co-principal investigator with John Hennessy on Joint Services Electronics Program contract DAAG29-79-C-0047 entitled "Reliability in Distributed Systems."

Susan Owicki was a member of the program committee for the International Symposium on the Semantics of Concurrent Computation, Evian, France, July, 1979.

Susan Owicki is on Computer Science and Technology Board of the National Research Council Assembly of Mathematical and Physical Sciences.